

SQL Basics

Joseph Machado

Run instructions

1. Create a [GitHub codespaces machine with this link](#).
2. Run `./setup.sh` in the terminal.
3. Open duckdb sql ide with `./duckdb tpch.db`.
4. Run the queries in this cheatsheet.

Data Model

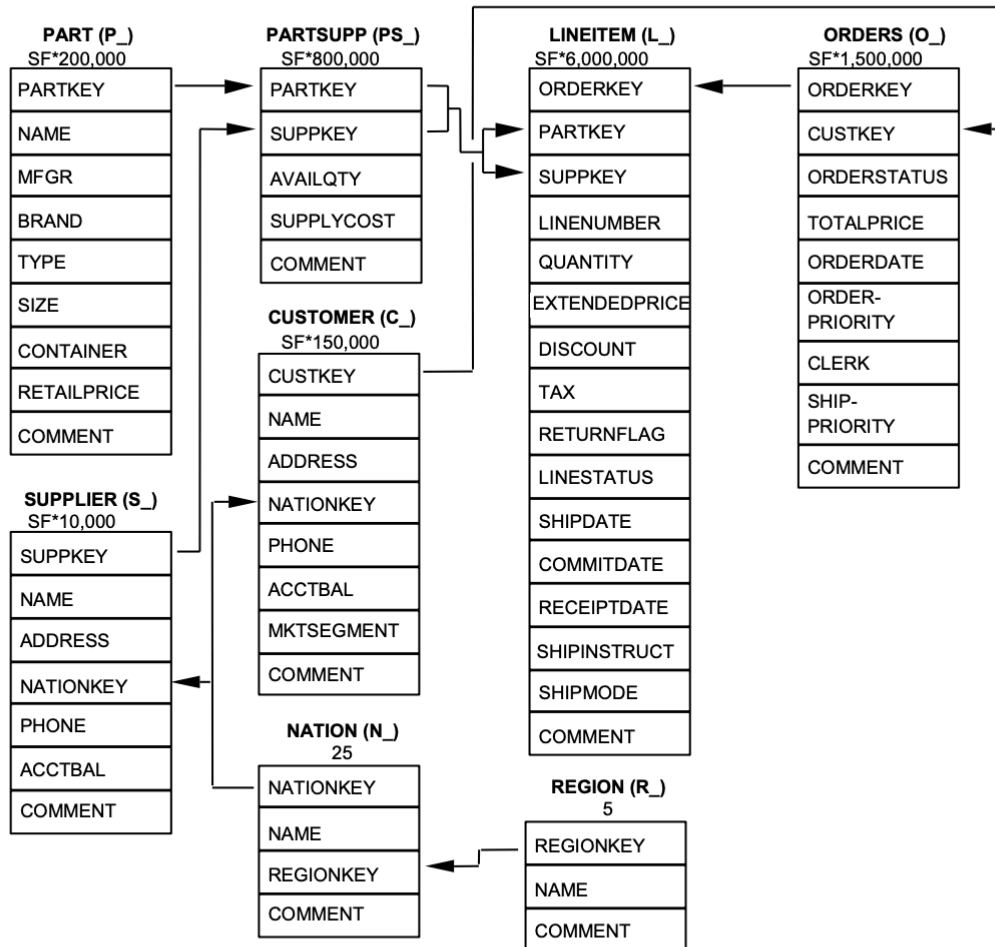
The TPC-H data represents a car parts seller's data warehouse, where we record orders, items that make up that order (lineitem), supplier, customer, part (parts sold), region, nation, and partsupp (parts supplier).

Note: Have a copy of the data model as you follow along; this will help in understanding the examples provided and in answering exercise questions.

1.2 Database Entities, Relationships, and Characteristics

The components of the TPC-H database are defined to consist of eight separate and individual tables (the Base Tables). The relationships between columns of these tables are illustrated in Figure 2: The TPC-H Schema.

Figure 2: The TPC-H Schema



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

Use SELECT...FROM, LIMIT, WHERE, & ORDER BY to read the required data from tables

The most common use for querying is to read data in our tables. We can do this using a SELECT ... FROM statement, as shown below.

```
-- use * to specify all columns
SELECT * FROM orders limit 5;

-- use column c_names only to read data from those columns
SELECT o_orderkey, o_totalprice FROM orders LIMIT 5;
```

However, running a SELECT ... FROM statement can cause issues when the data set is extensive.

```
SELECT o_orderkey, o_totalprice FROM orders LIMIT 5;
```

We can use the WHERE clause if we want to get the rows that match specific criteria. We can specify one or more filters within the WHERE clause.

The WHERE clause with more than one filter can use combinations of AND and OR criteria to combine the filter criteria, as shown below.

```
-- all customer rows that have c_nationkey = 20
SELECT * FROM customer WHERE c_nationkey = 20 LIMIT 10;

-- all customer rows that have c_nationkey = 20 and c_acctbal > 1000
SELECT * FROM customer
WHERE c_nationkey = 20 AND c_acctbal > 1000 LIMIT 10;

-- all customer rows that have c_nationkey = 20 or c_acctbal > 1000
SELECT * FROM customer
WHERE c_nationkey = 20 OR c_acctbal > 1000 LIMIT 10;

-- all customer rows that have (c_nationkey = 20 and c_acctbal > 1000) or rows that have c_n
SELECT * FROM customer
WHERE (c_nationkey = 20 AND c_acctbal > 1000)
OR c_nationkey = 11 LIMIT 10;
```

We can combine multiple filter clauses, as seen above. We have seen examples of equals (=) and greater than (>) conditional operators. There are 6 **conditional operators**, they are

1. < Less than
2. > Greater than
3. <= Less than or equal to
4. >= Greater than or equal to
5. = Equal
6. <> and != both represent Not equal (some DBs only support one of these)

Additionally, for string types, we can make **pattern matching with like condition**. In a **like** condition, a **_** means any single character, and **%** means zero or more characters, for example.

```
-- all customer rows where the c_name has a 381 in it
SELECT * FROM customer WHERE c_name LIKE '%381%';

-- all customer rows where the c_name ends with a 381
SELECT * FROM customer WHERE c_name LIKE '%381';

-- all customer rows where the c_name starts with a 381
SELECT * FROM customer WHERE c_name LIKE '381%';

-- all customer rows where the c_name has a combination of any character and 9 and 1
SELECT * FROM customer WHERE c_name LIKE '%_91%';
```

We can also filter for more than one value using **IN** and **NOT IN**.

```
-- all customer rows which have c_nationkey = 10 or c_nationkey = 20
SELECT * FROM customer WHERE c_nationkey IN (10,20);

-- all customer rows which have do not have c_nationkey as 10 or 20
SELECT * FROM customer WHERE c_nationkey NOT IN (10,20);
```

We can get the number of rows in a table using **count(*)** as shown below.

```
SELECT COUNT(*) FROM customer;
SELECT COUNT(*) FROM lineitem;
```

If we want to get the rows sorted by values in a specific column, we use **ORDER BY**, for example.

```
-- Will show the first ten customer records with the lowest custkey
-- rows are ordered in ASC order by default
SELECT * FROM orders ORDER BY o_custkey LIMIT 10;

-- Will show the first ten customer's records with the highest custkey
SELECT * FROM orders ORDER BY o_custkey DESC LIMIT 10;
```

Combine data from multiple tables using JOINS (there are different types of JOINS)

We can combine data from multiple tables using joins. When we write a join query, we have a format as shown below.

```
-- not based on real tables
SELECT
    a.*
FROM
    table_a a -- LEFT table a
    JOIN table_b b -- RIGHT table b
    ON a.id = b.id
```

The table specified first (table_a) is the left table, whereas the table established second is the right table. When we have multiple tables joined, we consider the joined dataset from the first two tables as the left table and the third table as the right table (The DB optimizes the joins for performance).

```
-- not based on real tables
SELECT
    a.*
FROM
    table_a a -- LEFT table a
    JOIN table_b b -- RIGHT table b
    ON a.id = b.id
    JOIN table_c c -- LEFT table is the joined data from table_a & table_b, right table is table_c
    ON a.c_id = c.id
```

There are five main types of joins, they are:

1. Inner join (default): Get only rows in both tables

```
SELECT
    o.o_orderkey,
    l.l_orderkey
FROM
    orders o
    JOIN lineitem l ON o.o_orderkey = l.l_orderkey
    AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY
    AND l.l_shipdate + INTERVAL '5' DAY
LIMIT
    100;

SELECT
    COUNT(o.o_orderkey) AS order_rows_count,
    COUNT(l.l_orderkey) AS lineitem_rows_count
FROM
    orders o
    JOIN lineitem l ON o.o_orderkey = l.l_orderkey
    AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY
    AND l.l_shipdate + INTERVAL '5' DAY;
```

Note: JOIN defaults to INNER JOIN.

The output will have rows from orders and lineitem that found at least one matching row from the other table with the specified join condition (same o_orderkey and orderdate within ship date +/- 5 days).

2. Left outer join (aka left join): Get all rows from the left table and only matching rows from the right table.

```
SELECT
    o.o_orderkey,
    l.l_orderkey
FROM
    orders o
    LEFT JOIN lineitem l ON o.o_orderkey = l.l_orderkey
    AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY
    AND l.l_shipdate + INTERVAL '5' DAY
LIMIT
```

```

100;

SELECT
    COUNT(o.o_orderkey) AS order_rows_count,
    COUNT(l.l_orderkey) AS lineitem_rows_count
FROM
    orders o
    LEFT JOIN lineitem l ON o.o_orderkey = l.l_orderkey
    AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY
    AND l.l_shipdate + INTERVAL '5' DAY;

```

The output will have all the rows from orders and the rows from lineitem that were able to find at least one matching row from the orders table with the specified join condition (same o_orderkey and orderdate within ship date +/- 5 days).

3. Right outer join (aka right join): Get matching rows from the left and all rows from the right table.

```

SELECT
    o.o_orderkey,
    l.l_orderkey
FROM
    orders o
    LEFT JOIN lineitem l ON o.o_orderkey = l.l_orderkey
    AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY
    AND l.l_shipdate + INTERVAL '5' DAY
LIMIT
    100;

SELECT
    COUNT(o.o_orderkey) AS order_rows_count,
    COUNT(l.l_orderkey) AS lineitem_rows_count
FROM
    orders o
    JOIN lineitem l ON o.o_orderkey = l.l_orderkey
    AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY
    AND l.l_shipdate + INTERVAL '5' DAY;

```

The output will have the rows from orders that found at least one matching row from the lineitem table with the specified join condition (same o_orderkey and orderdate within ship date +/- 5 days) and all the rows from the lineitem table.

4. Full outer join: Get all rows from both the left and right tables.

```
SELECT
    o.o_orderkey,
    l.l_orderkey
FROM
    orders o
    LEFT JOIN lineitem l ON o.o_orderkey = l.l_orderkey
    AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY
    AND l.l_shipdate + INTERVAL '5' DAY
LIMIT
    100;

SELECT
    COUNT(o.o_orderkey) AS order_rows_count,
    COUNT(l.l_orderkey) AS lineitem_rows_count
FROM
    orders o
    FULL OUTER JOIN lineitem l ON o.o_orderkey = l.l_orderkey
    AND o.o_orderdate BETWEEN l.l_shipdate - INTERVAL '5' DAY
    AND l.l_shipdate + INTERVAL '5' DAY;
```

The output will have all the rows from orders that found at least one matching row from the lineitem table with the specified join condition (same o_orderkey and orderdate within ship date +/- 5 days) and all the rows from the lineitem table.

5. Cross join: Get the cartesian product of all rows

```
SELECT
    n.n_name AS nation_c_name,
    r.r_name AS region_c_name
FROM
    nation n
    CROSS JOIN region r;
```

The output will have every row of the nation joined with every row of the region. There are 25 nations and five regions, leading to 125 rows in our result from the cross-join.

There are cases where we will need to join a table with itself, called a SELF-join.

Example:

1. For every customer order, get the order placed earlier in the same week (Sunday - Saturday, not the previous seven days). Only show customer orders that have at least one such order.

```
SELECT
    o1.o_custkey
FROM
    orders o1
JOIN orders o2 ON o1.o_custkey = o2.o_custkey
AND year(o1.o_orderdate) = year(o2.o_orderdate)
AND week(o1.o_orderdate) = week(o2.o_orderdate)
WHERE
    o1.o_orderkey != o2.o_orderkey;
```

Generate metrics for your dimension(s) using GROUP BY

Most analytical queries require calculating metrics that involve combining data from multiple rows. **GROUP BY** allows us to perform aggregate calculations on data from a set of rows grouped together based on values of specified column(s).

Example:

1. Create a report that shows the number of orders per orderpriority segment.

```
SELECT
    o_orderpriority,
    count(*) AS num_orders
FROM
    orders
GROUP BY
    o_orderpriority;
```

In the above query, we group the data by **orderpriority**, and the calculation **count(*)** will be applied to the rows having a specific **orderpriority** value. The output will consist of one row per unique value of **orderpriority** and the **count(*)** calculation.

The calculations allowed are typically **SUM/MIN/MAX/AVG/COUNT**. However, some databases have more complex aggregate functions; check your DB documentation.

Use the result of a query within a query using sub-queries

When we want to use the result of a query as a table in another query, we use subqueries.

Example:

1. Create a report that shows the nation, how many items it supplied (by suppliers in that nation), and how many items it purchased (by customers in that nation).

```
SELECT
    n.n_name AS nation_c_name,
    s.quantity AS supplied_items_quantity,
    c.quantity AS purchased_items_quantity
FROM
    nation n
    LEFT JOIN (
        SELECT
            n.n_nationkey,
            sum(l.l_quantity) AS quantity
        FROM
            lineitem l
            JOIN supplier s ON l.l_suppkey = s.s_suppkey
            JOIN nation n ON s.s_nationkey = n.n_nationkey
        GROUP BY
            n.n_nationkey
    ) s ON n.n_nationkey = s.n_nationkey
    LEFT JOIN (
        SELECT
            n.n_nationkey,
            sum(l.l_quantity) AS quantity
        FROM
            lineitem l
            JOIN orders o ON l.l_orderkey = o.o_orderkey
            JOIN customer c ON o.o_custkey = c.c_custkey
            JOIN nation n ON c.c_nationkey = n.n_nationkey
        GROUP BY
            n.n_nationkey
    ) c ON n.n_nationkey = c.n_nationkey;
```

In the above query, we can see that there are two sub-queries, one to calculate the quantity supplied by a nation and the other to calculate the quantity purchased by the customers of a nation.

Replicate IF.ELSE logic with CASE statements

We can do conditional logic in the `SELECT ... FROM` part of our query, as shown below.

```
SELECT
  o_orderkey,
  o_totalprice,
  CASE
    WHEN o_totalprice > 100000 THEN 'high'
    WHEN o_totalprice BETWEEN 25000
      AND 100000 THEN 'medium'
    ELSE 'low'
  END AS order_price_bucket
FROM
  orders;
```

We can see how we display different values depending on the `totalprice` column. We can also use multiple criteria as our conditional criteria (e.g., `totalprice > 100000 AND orderpriority = '2-HIGH'`).

Use standard inbuilt DB functions for common String, Time, and Numeric data manipulation

When processing data, more often than not, we will need to change values in columns; shown below are a few standard functions to be aware of:

1. String functions

1. **LENGTH** is used to calculate the length of a string. E.g., `SELECT LENGTH('hi');` will output 2.
2. **CONCAT** combines multiple string columns into one. E.g., `SELECT CONCAT(o_orderstatus, '-', o_orderpriority) FROM ORDERS LIMIT 5;` will concatenate the `o_orderstatus` and `o_orderpriority` columns with a dash in between them.
3. **SPLIT** is used to split a value into an array based on a given delimiter. E.g., `SELECT STRING_SPLIT(o_orderpriority, '-') FROM ORDERS LIMIT 5;` will output a column with arrays formed by splitting `o_orderpriority` values on `-`.
4. **SUBSTRING** is used to get a sub-string from a value, given the start and end character indices. E.g., `SELECT o_orderpriority, SUBSTRING(o_orderpriority, 1, 5) FROM orders LIMIT 5;` will get the first five (1 - 5) characters of the `o_orderpriority` column.

5. **TRIM** is used to remove empty spaces to the left and right of the value. E.g., `SELECT TRIM(' hi ');` will output `hi` without any spaces around it. `LTRIM` and `RTRIM` are similar but only remove spaces before and after the string, respectively.

2. Date and Time functions

1. **Adding and subtracting dates:** Is used to add and subtract periods; the format heavily depends on the DB. In DuckDB, `date_diff` accepts 3 parameters, the outputs unit (day, month, year), the datetime/date values a and b such that the output will be a - b. The `+ INTERVAL n UNIT(DAY/MONTH/YEAR)` adds the value (in specified units) to the timestamp value.

```
-- Date and Time Functions
SELECT
    date_diff('day', DATE '2022-10-01', DATE '2023-11-05') AS diff_in_days,
    date_diff('month', DATE '2022-10-01', DATE '2023-11-05') AS diff_in_months,
    date_diff('year', DATE '2022-10-01', DATE '2023-11-05') AS diff_in_years,
    DATE '2022-10-01' + INTERVAL 400 DAY AS new_date;
```

It will show the difference between the two dates in the specified period. We can also add/subtract an arbitrary period from a date/time column. E.g., `SELECT DATE '2022-11-05' + INTERVAL '10' DAY;` will show the output `2022-11-15` (try subtraction of dates).

2. **String \Leftrightarrow date/time conversions:** When we want to change the data type of a string to date/time, we can use the `DATE 'YYYY-MM-DD'` or `TIMESTAMP 'YYYY-MM-DD HH:mm:ss'` functions. But when the data is in a non-standard date/time format such as `MM/DD/YYYY`, we will need to specify the input structure; we do this using `strptime`, E.g., `SELECT strptime('11-05-2023', '%m-%d-%Y');`.

We can convert a timestamp/date into a string with the required format using `strftime`. E.g., `SELECT strftime(o_orderdate, '%Y-%m-01') AS first_month_date FROM orders LIMIT 5;` will map every `o_orderdate` to the first of their month.

See [this page](#) on how to set the proper date time format.

3. **Time frame functions (YEAR/MONTH/DAY):** When we want to extract specific periods from a date/time column, we can use these functions. E.g., `SELECT year(DATE '2023-11-05');` will return `2023`. Similarly, we have month, day, hour, min, etc.

3. Numeric

1. **ROUND** is used to specify the number of digits allowed after the decimal point. E.g. `SELECT ROUND(100.102345, 2);`

2. **ABS** is used to get the absolute value of a given number. E.g. `SELECT ABS(-100), ABS(100);`
3. **Mathematical operations** these are `+`, `-`, `*`, `/`.
4. **Ceil/Floor** is used to get the next higher and most recent lower integers, given a decimal digit. E.g. `SELECT CEIL(100.1), FLOOR(100.1);`